3. Concurrency in Java and its memory model

Nelma Moreira & **José Proença**

Concurrent programming (CC3040) 2023/2024

CISTER – U.Porto, Porto, Portugal

`https://fm-dcc.github.io/pc2324`

U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

CENTRO DE MATEMÁTICA
UNIVERSIDADE DO PORTO

CISTER - Research Centre in Real-Time & Embedded Computing Systems

Overview

We are here

Blocks of sequential code running concurrently and sharing memory:

- What is Scala?
- Concurrency in Java and its memory model
- Basic concurrency blocks and libraries
- Futures and promises
- Actor model (maybe)

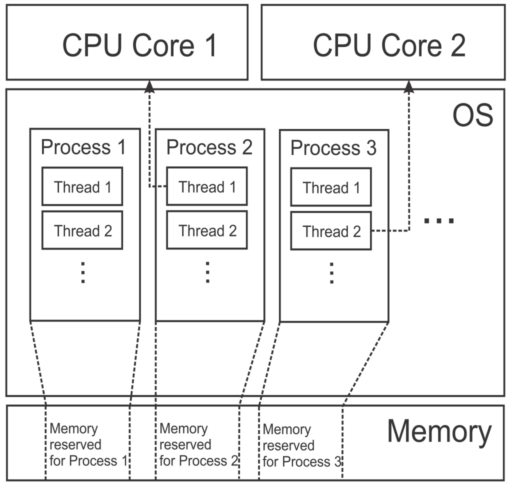
Traditional concurrency

Synchronisation

- Coordination of multiple executions in a concurrent system
- Mechanisms to order concurrent executions
- Mechanisms to exchange information

Exchanging information

- Concurrent programs: shared memory communication
- Distributed programs: message passing communication



[in *"Learning Concurrent Programming in Scala"*, pg. 32]

Starting a new JVM instance always creates only one process.

In that process, multiple threads can run simultaneously.

Unlike runtimes (e.g. Python), the JVM:

- does not implement its custom threads,
- maps each Java thread to an OS thread

Managing threads

Current thread

```scala
object ThreadsMain extends App {
  val t: Thread =
      Thread.currentThread
  val name = t.getName
  println(s"I am the thread $name")
}
```

Using SBT, this prints:

`[info] I am the thread sbt-bg-threads-1`

In SBT do “`set fork := true`”

It will then it prints:

`[info] I am the thread main`

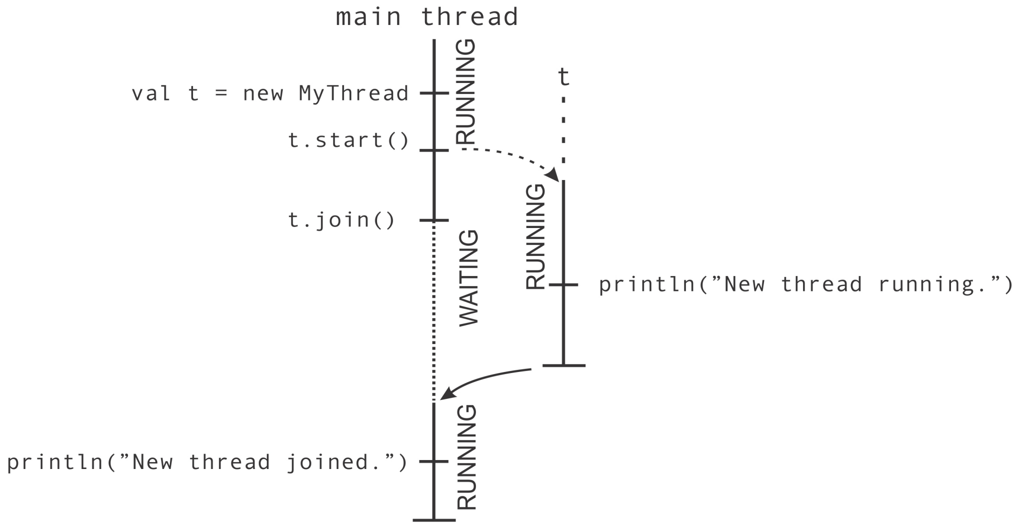
Creating threads

```scala
object ThreadsCreation extends App {
  class MyThread extends Thread {
    override def run(): Unit = {
      println("New thread running.")
    }
  }
  val t = new MyThread
  t.start()
  t.join()
  println("New thread joined.")
}
```

`start` eventually causes `run` to execute in a new thread;

the OS decides when;

`join` puts the main thread in a waiting state, and allows the OS to re-assign the processor.



[in *"Learning Concurrent Programming in Scala"*, pg. 35]

Simpler thread creation

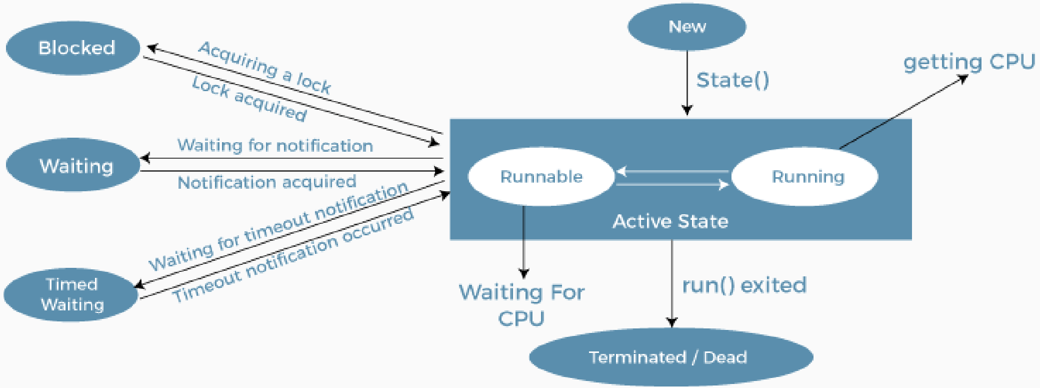
```scala
def thread(body: =>Unit): Thread = {
  val t = new Thread {
    override def run() = body
  }
  t.start()
  t
}
```

Simpler thread creation

```
def thread(body: =>Unit): Thread = {
  val t = new Thread {
    override def run() = body
  }
  t.start()
  t
}
```

Using the thread function

```
object ThreadsSleep extends App {
  val t = thread {
    Thread.sleep(1000)
    log("New thread running.")
    Thread.sleep(1000)
    log("Still running.")
    Thread.sleep(1000)
    log("Completed.")
  }
  t.join()
  log("New thread joined.")
}
```



Life Cycle of a Thread

[in https://static.javatpoint.com/core/images/life-cycle-of-a-thread.png]

Nondeterministic thread execution

- `"New thread"` printed always at the end
- Other prints not always in the same order – nondeterministic execution
- Common in concurrent applications – what makes it so hard
- Note: `join` also forces all memory writes from the threads before proceeding

```
object ThreadsNondeterminism
    extends App {
 val t = thread { log("New␣thread␣
     running.") }
     log("...")
     log("...")
     t.join()
     log("New␣thread␣joined.")
}
```

Control of the execution order

- `join` provides guarantees that other threads terminated
- Not enough – we may want to inform other treads without terminating

Example 1: shared counter for unique IDs

```
object ThreadsUnprotectedUid extends App {
  var uidCount = 0L
  def getUniqueId() = {
    val freshUid = uidCount + 1
    uidCount = freshUid
    freshUid
}
```

What can go wrong?

```scala
...
def printUniqueIds(n: Int): Unit = {
  val uids = for (i<- 0 until n)
      yield getUniqueId()
  log(s"Generated␣uids:␣$uids")
}
val t = thread { printUniqueIds(5) }
printUniqueIds(5)
t.join()
...
```

```scala
object ThreadsNondeterminism
    extends App {
 val t = thread { log("New␣thread␣
     running.") }
     log("...")
     log("...")
     t.join()
     log("New␣thread␣joined.")
}
```

What do you expect?

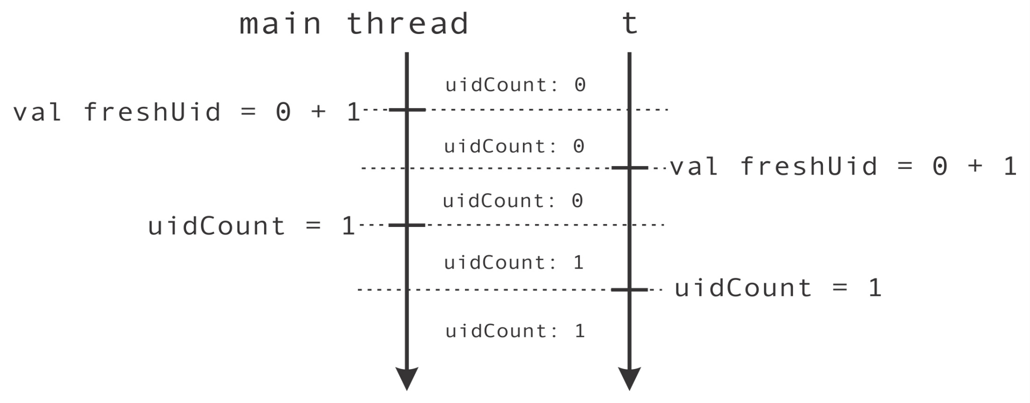
```scala
...
def printUniqueIds(n: Int): Unit = {
  val uids = for (i<- 0 until n)
      yield getUniqueId()
  log(s"Generated␣uids:␣$uids")
}
val t = thread { printUniqueIds(5) }
printUniqueIds(5)
t.join()
...
```

```scala
object ThreadsNondeterminism
    extends App {
 val t = thread { log("New␣thread␣
     running.") }
     log("...")
     log("...")
     t.join()
     log("New␣thread␣joined.")
}
```

Race Condition

when the output of a concurrent program depends on how the statements are scheduled.

`val freshUid = uidCount + 1 ; uidCount = freshUid ; freshUid`



[in *"Learning Concurrent Programming in Scala"*, pg. 40]

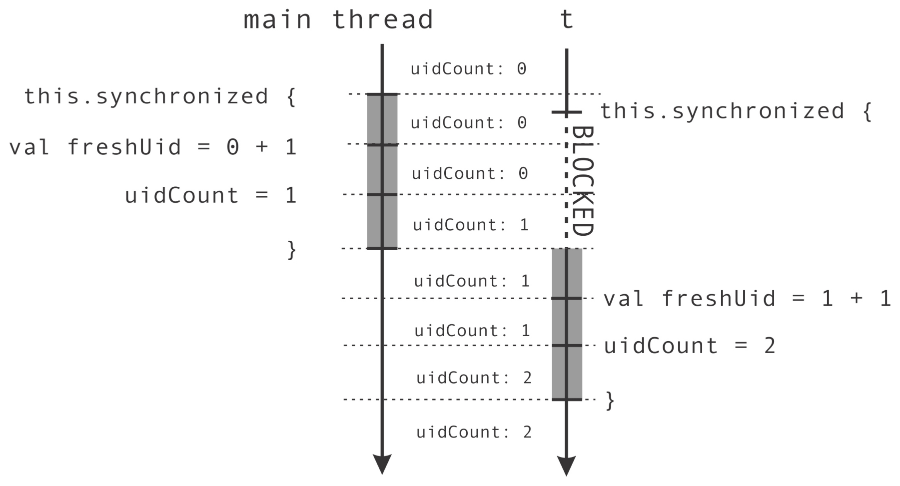
“Synchronized” to the rescue

```
def getUniqueId() =
  this.synchronized {
    val freshUid = uidCount + 1
    uidCount = freshUid
    freshUid
}
```

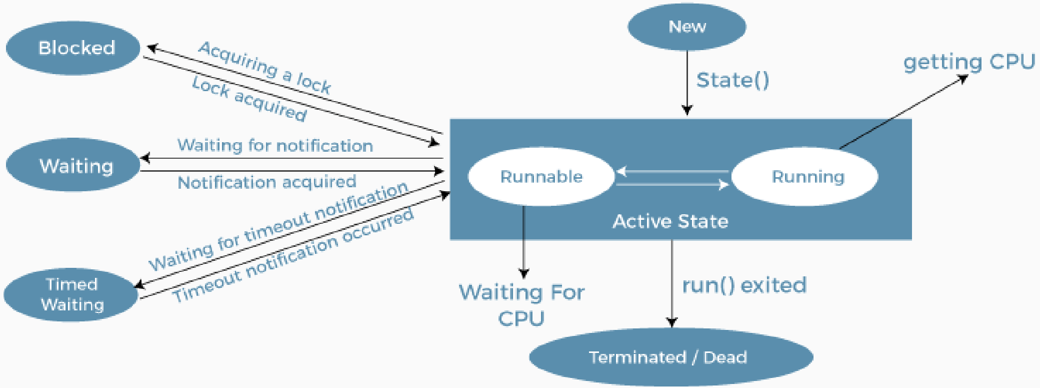
`synchronized` is:

- a fundamental Scala/Java construct for atomic executions
- can be called in any object (or instance of a class)
- ensures atomic execution wrt the object
- we say `obj.synchronized`
  - acquires the lock/monitor of obj at the start
  - releases the lock/monitor of obj at the end

Updating counter in parallel atomically



[in *"Learning Concurrent Programming in Scala"*, pg. 41]



Life Cycle of a Thread

[in https://static.javatpoint.com/core/images/life-cycle-of-a-thread.png]

Reordering

- using the `synchronized` statement has some (not too large) overhead
- not using `synchronized` can easily lead to errors, even if all seems correct

Find the bug in the next slide...

Find the bug

```scala
object ThreadSharedStateAccessReordering extends App {
  for (i <- 0 until 100000) {
    var a = false
    var b = false
    var x = -1
    var y = -1
    val t1 = thread {
      a = true
      y = if (b) 0 else 1
    }
    val t2 = thread {
      b = true
      x = if (a) 0 else 1
    }
    t1.join()
    t2.join()
    assert(!(x==1 && y==1), s"x=$x, y=$y")
  }
}
```

Reordering within threads

- The previous code can raise an error: both `x` and `y` can become 1!
- JVM can reorder statements in a thread when they seem to be independent.
- Because some processors do not always execute instructions in the expected order, to increase performance.
- (Known as "weak memory model")
- A `synchronized` block would solve this:
  - also enclosing each assignment in a `synchronized` block
  - `synchronized` sets up a *memory barrier*

- every object has a *lock*
- a running thread can aquire multiple locks from different objects

Example 2: Logging Bank Transfers

```
object SynchronizedNesting extends App {
  import scala.collection._

  private val transfers = mutable.ArrayBuffer[String]()
  def logTransfer(name: String, n: Int) = transfers.synchronized {
    transfers += s"transfer to account '$name' = $n"
  }
  class Account(val name: String, var money: Int)
  def add(account: Account, n: Int) = account.synchronized {
      account.money += n
      if (n > 10) logTransfer(account.name, n)
  }
  ...
}
```

```scala
private val transfers = mutable.ArrayBuffer[String]()
def logTransfer(name: String, n: Int) = transfers.synchronized {
  transfers += s"transfer to account '$name' = $n"
}
class Account(val name: String, var money: Int)
def add(account: Account, n: Int) = account.synchronized {
    account.money += n
    if (n > 10) logTransfer(account.name, n)
}

val jane = new Account("Jane", 100)
val john = new Account("John", 200)
val t1 = thread { add(jane, 5) }
val t2 = thread { add(john, 50) }
val t3 = thread { add(jane, 70) } // will not corrupt Jane's account
t1.join(); t2.join(); t3.join()
log(s"--- transfers ---\n$transfers")
```

Deadlocks

Deadlock

when two or more executions wait for each other before proceeding

- Studied in the first module with prof. Nelma Moreira
- Dining philosophers is a typical example
- Often caused by locks that are not released at the right time

```scala
object SynchronizedDeadlock extends App {
  import SynchronizedNesting.Account
  def send(a: Account, b: Account, n: Int) = a.synchronized {
    b.synchronized {
      a.money -= n
      b.money += n
    }
  }
  ... // can this go wrong?
}
```

Deadlocks – the dark side of locks

```scala
def send(a: Account, b: Account, n: Int) = a.synchronized {
  b.synchronized {
    a.money -= n
    b.money += n
  }
}

val a = new Account("Jack", 1000)
val b = new Account("Jill", 2000)
val t1 = thread { for (i<- 0 until 100) send(a, b, 1) }
val t2 = thread { for (i<- 0 until 100) send(b, a, 1) }
t1.join(); t2.join()
log(s"a␣=␣a.money, b ={b.money}")
```

Deadlocks – the dark side of locks

```scala
def send(a: Account, b: Account, n: Int) = a.synchronized {
  b.synchronized {
    a.money -= n
    b.money += n
  }
}

val a = new Account("Jack", 1000)
val b = new Account("Jill", 2000)
val t1 = thread { for (i<- 0 until 100) send(a, b, 1) }
val t2 = thread { for (i<- 0 until 100) send(b, a, 1) }
t1.join(); t2.join()
log(s"a␣=␣a.money, b ={b.money}")
```

It works but...

```
def send(a: Account, b: Account, n: Int) = a.synchronized {
  b.synchronized {
    a.money -= n
    b.money += n
  }
}

val a = new Account("Jack", 1000)
val b = new Account("Jill", 2000)
val t1 = thread { for (i<- 0 until 100) send(a, b, 1) }
val t2 = thread { for (i<- 0 until 100) send(b, a, 1) }
t1.join(); t2.join()
log(s"a␣=␣a.money, b ={b.money}")
```

It works but... it can deadlock

Possible fix: fix order

- always acquire locks in the same order
- need a total order on locks
- we can use the getUniqueId (Example 1)

```
import SynchronizedProtectedUid.getUniqueId
class Account(val name: String, var money: Int) {
  val uid = getUniqueId()
}
```

Possible fix: fix order

- always acquire locks in the same order
- need a total order on locks
- we can use the getUniqueId (Example 1)

```
import SynchronizedProtectedUid.getUniqueId
class Account(val name: String, var money: Int) {
  val uid = getUniqueId()
}
```

```
def send(a1: Account, a2: Account, n: Int) {
  def adjust() {
    a1.money -= n
    a2.money += n
  }
  if (a1.uid < a2.uid)  a1.synchronized{ a2.synchronized{ adjust() }}
  else                  a2.synchronized{ a1.synchronized{ adjust() }}
}
```

Guarded blocks

Guarded block (for us)

a block of code that waits for a condition before running in a thread

Example 3: Thread pool with a queue of tasks

- Creating new threads in Java is expensive and avoidable
- Usually we re-use threads, by maintaining a set of waiting threads
- This set is call a thread pool
  - Scala already provides thread pools
  - We first create our own

```scala
import scala.collection._
object SynchronizedBadPool extends App {
 // our set of tasks
 private val tasks = mutable.Queue[()=>Unit]()

 // our single working thread
 val worker = new Thread {
   def poll(): Option[()=>Unit] =
     asks.synchronized {
      if (tasks.nonEmpty) Some(tasks.dequeue())
      else                None
     }
   // keep on trying to run forever!
   override def run() = while (true)
     poll() match {
       case Some(task) => task()
       case None =>
   }
 }
```

```scala
  // starting the worker as
     a daemon
  worker.setName("Worker")
  worker.setDaemon(true)
  worker.start()

  def asynchr(body: =>Unit) =
    tasks.synchronized {
      tasks.enqueue(()=>body)
    }

  asynchr{ log("Hello")  }
  asynchr{ log("␣world!")}
  Thread.sleep(5000)
}
```

Note on daemon threads

Normal

- not the default
- have lower priority
- terminated automatically when JVM terminates
- in other words, do not prevent the JVM from terminating
- (the JVM terminates when 'normal' tasks terminate)

Busy-waiting is bad

- needlessly uses processor power (and drains the battery)
- after executing the previous code the `worker` will keep on running (unless you set in SBT `set fork := true`,)
- in general, we want the `worker` to enter a waiting state

synchronized + wait + notify

- these are methods that every Java/Scala object has
- wait:
  - needs the lock
  - puts the thread in a waiting state
  - releases the lock until activation
- notify:
  - needs the lock
  - activates all waiting threads
- Note that the JVM can decide to call `wait` on its own – spurious wakeups – needing to re-enter the *wait*

Wait-notify example

```
object SynchronizedGuardedBlocks extends App {
  val lock = new AnyRef
  var message: Option[String] = None
  val greeter = thread {
    lock.synchronized {
      while (message == None) lock.wait() // non-busy waiting for a message
      log(message.get)                    // it will eventually log!
    }
  }
  lock.synchronized {
    message = Some("Hello!")
    lock.notify()                         // awakes the (possibly) locked thread
  }
  greeter.join()
}
```

Example 3 – without busy-waiting

```scala
import scala.collection._
object SynchronizedPool extends App {
  private val tasks = mutable.Queue[()=>Unit]()

  object Worker extends Thread {
    setDaemon(true)
    def poll() = tasks.synchronized {
      while (tasks.isEmpty) tasks.wait()
                                              // now using wait
      tasks.dequeue()
    }
    override def run() = while (true) {
      val task = poll()
      task()
    }
  }
```

```scala
  Worker.start()

  def asynchr(body: =>Unit) =
    tasks.synchronized {
       tasks.enqueue(()=>body)
       // now notifying
       tasks.notify()
    }

  asynchr{ log("Hello")  }
  asynchr{ log(" world!")}
  Thread.sleep(500)
}
```

Interrupting threads – `Thread.interrupt()`

- Our `Worker` can run forever (`while-true`)
- Terminates when the JVM terminates (daemon)
- `Worker` can be terminated earlier while waiting
  - `Worker.interrupt()`
  - triggers an `InterruptedException` that can be handled
  - if it was not waiting, then no exception is raised
  - instead a flag `Worker.isInterrupted` becomes true

Interrupting threads – alternative with graceful shutdown

```scala
object Worker extends Thread {
  var terminated = false
  // "manually" terminate when asked
  def poll(): Option[() => Unit] = tasks.synchronized {
    while (tasks.isEmpty && !terminated) tasks.wait()
    if (!terminated) Some(tasks.dequeue()) else None
  }

  import scala.annotation.tailrec
  @tailrec override def run() = poll() match {
    case Some(task) => task(); run()
    case None =>
  }
  // "manually" ask to terminate
  def shutdown() = tasks.synchronized {
    terminated = true
    tasks.notify()
  }
}
```

Volatile variables – Alternative to `lock.synchronized`

- using the `@volatile` annotation

- can be atomically read and modified
- mostly used as status flag
- are never reordered in a thread
- writes are immediately visible to other threads
- very cheap to read
- not enough in many situations (e.g., `getUniqueID`)
- enough for previous example – Slide 18

Example 4 – Batman

```
class Page(val txt: String, var position: Int)

object Volatile extends App {
  val pages = for (i<- 1 to 5) yield
    new Page("Na" * (100 - 20 * i) + "␣Batman!", -1)
  @volatile var found = false
  for (p <- pages) yield thread {
    var i = 0
    while (i < p.txt.length && !found)
      if (p.txt(i) == '!') {
        p.position = i
        found = true
      } else i += 1
  }
  while (!found) {}
  log(s"results:␣${pages.map(_.position)}")
}
```

The Java Memory Model overview

Happens-before relation

action α happens-before action β

means action β sees the memory writes of action α

- **Program order:** α in a thread HB every subsequent β in that program and thread
- **Monitor locking:** unlocking HB every subsequent locking (of the same lock)
- **Volatile fields:** writing to a volatile field HB every of its subsequent read
- **Thread start:** calling `thrd.start()` HB any actions of `thrd`
- **Thread termination:** α in a thread HB a `join()` on that thread.
- **Transitivity:** if α HB β and β HB γ, then aα HB γ

Happens-before relation

action α happens-before action β
means action β sees the memory writes of action α

- **Program order:** α in a thread HB every subsequent β in that program and thread
- **Monitor locking:** unlocking HB every subsequent locking (of the same lock)
- **Volatile fields:** writing to a volatile field HB every of its subsequent read
- **Thread start:** calling `thrd.start()` HB any actions of `thrd`
- **Thread termination:** α in a thread HB a `join()` on that thread.
- **Transitivity:** if α HB β and β HB γ, then aα HB γ

Data race: when a write to memory does not happen-before its intended read.

Immutable objects and final fields

```
class Foo(
  final val a: Int, val b:
      Int, c: Int)

class Foo { // Java code below
  final private int a$;
  final private int b$;
  final private int c$;
  final public int a() {
    return a$; }
  public int b() { return b$; }
  public Foo(int a,
             int b,
             int c) {
    a$ = a; b$ = b; c$ = c;
  }
}
```

- Final fields: cannot be overridden
- val: cannot be updated
- vals are final
- Some collections are immutable (e.g. `List`), but contain non-final fields
  - need synchronisation when shared
- Objects with only non-final fields
  - do not need synchronisation when shared

Summary of operators

- synchronize
- wait
- notify
- ...